# The Path-Indexing Method for Indexing Terms

Technical Note 473

October 1989

By: Mark E. Stickel, Sr. Computer Scientist

Artificial Intelligence Center
Computer and Information Sciences Division

This research is supported by the Defense Advanced Research Projects Agency under Contract N00039-88-C-0248 with the Space and Naval Warfare Systems Command. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States government. APPROVED FOR PUBLIC RELEASE. DISTRIBUTION UNLIMITED.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

# Report Documentation Page

*Form Approved*
*OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

| | | |
|---|---|---|
| 1. REPORT DATE<br>**OCT 1989** | 2. REPORT TYPE | 3. DATES COVERED<br>**00-10-1989 to 00-10-1989** |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>**The Path-Indexing Method for Indexing Terms** | 5a. CONTRACT NUMBER<br>5b. GRANT NUMBER<br>5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | 5d. PROJECT NUMBER<br>5e. TASK NUMBER<br>5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>**SRI International,333 Ravenswood Avenue,Menlo Park,CA,94025** | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSOR/MONITOR’S ACRONYM(S)<br>11. SPONSOR/MONITOR’S REPORT NUMBER(S) |

12. DISTRIBUTION/AVAILABILITY STATEMENT
**Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | **27** | |
| **unclassified** | **unclassified** | **unclassified** | | | |

## Abstract

The path-indexing method for indexing first-order predicate calculus terms is a refinement of the standard coordinate-indexing method. Path indexing offers much faster retrieval at a modest cost in space. Path indexing is compared with discrimination-net and codeword indexing. While discrimination-net indexing may often be the preferred method for maximum speed, path indexing is an effective alternative if discrimination-net indexing requires too much space or in certain cases in which discrimination-net indexing performs particularly poorly.

## 1 Introduction

Artificial intelligence (AI), logic programming, and automated deduction systems are often required to deal with large amounts of symbolic information. The need to store large amounts of information is met in conventional applications by database systems, but the form of the data in AI, logic programming, and automated deduction applications requires somewhat different techniques, particularly in the context of indexing the data for effective retrieval.

It is necessary in these applications to retrieve entries that are indexed by values that are at least as general as first-order predicate calculus terms. A first-order predicate calculus term is either (1) a constant, (2) a variable, or (3) an $n$-ary function symbol applied to $n$ term arguments. Examples are the constant terms *John*, *widget*, and 3.1416; the variable terms $x$ and $y$; and the composite terms $(a + x) - 3$, $father(John)$, and $append(nil, x, x)$. Conventional databases can easily index only constant terms, e.g., numbers and strings.

Besides retrieving exact matches or accepting any value as in conventional database retrieval operations, it is necessary, for example, to be able to specify retrieval of all stored terms that are unifiable with $x + (-x)$.

Terms can be related in several ways. A pair of terms can be *equal*, or equal except for the names of the variables, i.e., the terms are *variants*. One term can be an *instance* of another—the first is equal to the second with terms substituted for its variables. Conversely, one term can be a *generalization* of another. Finally, the terms can be *unifiable*, i.e., have a common instance.

Retrieval on a field containing keys that are first-order predicate calculus terms may require finding terms in the database that are

- Equal to the term in the retrieval request
- Variants of the term in the retrieval request
- Instances of the term in the retrieval request
- Generalizations of the term in the retrieval request
- Unifiable with the term in the retrieval request.

The need for each form of retrieval can be illustrated in the field of automated deduction [1,11]. Automated deduction systems often require all these forms of retrieval. Other applications, such as logic programming and expert systems, often use only a subset. To determine if a newly derived formula is already present in the database, it is necessary to retrieve terms that are equal to or variants of the new formula. It is necessary to find instances (resp., generalizations) of a newly derived formula to perform equality simplification or subsumption[1] of formulas in the database by the new formula (resp., of the new formula by formulas in the database). Other operations, such as resolution, may require unifiable terms. Prolog inference, as a special case of resolution, requires retrieval of clauses whose head literal is unifiable with the current goal.

The *path-indexing method* we propose is a refinement of the *coordinate-indexing method* that was proposed for the PLANNER AI programming language [4] and was used in the Logic Machine Architecture (LMA) [7] for implementing deduction systems (where it is called *FPA indexing*). Path indexing offers substantially faster retrieval in exchange for a modest increase in storage cost.

[1] In deduction, subsumption is the deletion of assertions that are instances of more general assertions. It is done to eliminate redundant information and reduce the size of the search space.

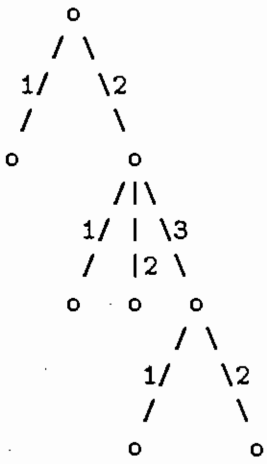

Figure 1: Tree corresponding to term $f(a, g(b, x, h(y, z)))$.

Section 2 contains a description of the simple and familiar coordinate-indexing method. Section 3 defines the new path-indexing method, which is easily understood by comparing it with the coordinate-indexing method. An implementation approach is given in Section 4. Path indexing is compared with coordinate indexing in Section 5 and compared with discrimination-net and codeword indexing methods in Section 6.

## 2 Coordinate Indexing

The term $t = f(a, g(b, x, h(y, z)))$ can be specified by a mapping $Symbol_t$ from the coordinates $\langle\rangle$, $\langle 1\rangle$, $\langle 2\rangle$, $\langle 2,1\rangle$, $\langle 2,2\rangle$, $\langle 2,3\rangle$, $\langle 2,3,1\rangle$, $\langle 2,3,2\rangle$ of the tree in Figure 1 to the symbols in $t$:

$$\begin{aligned}
Symbol_t(\langle\rangle) &= f \\
Symbol_t(\langle 1\rangle) &= a \\
Symbol_t(\langle 2\rangle) &= g \\
Symbol_t(\langle 2,1\rangle) &= b \\
Symbol_t(\langle 2,2\rangle) &= *^2 \\
Symbol_t(\langle 2,3\rangle) &= h \\
Symbol_t(\langle 2,3,1\rangle) &= * \\
Symbol_t(\langle 2,3,2\rangle) &= *
\end{aligned}$$

Let *Terms* be the set of terms stored for possible retrieval. Then the following can be used to define methods for retrieving those terms in *Terms* that are variants of, instances of, generalizations of, or unifiable with a term.[3] $GetVariants(\langle\rangle, t)$ returns a subset of *Terms* that includes all variants of $t$; $GetInstances(\langle\rangle, t)$ returns a subset that includes all instances; $GetGeneralizations(\langle\rangle, t)$ returns a subset that includes all generalizations; $GetUnifiables(\langle\rangle, t)$ returns a subset that includes all terms unifiable with $t$. If $t$ and *Terms* are all *linear* terms, i.e., do not contain repeated variables, then these retrieval operations are exact. If there are nonlinear terms, extra terms may be retrieved. The expression $p \cdot i$ denotes the coordinate $p$ extended by $i$, e.g., $\langle 2\rangle \cdot 3$ is $\langle 2, 3\rangle$.

Let $GetTerms(p, s) = \{t \in Terms | Symbol_t(p) = s\}$. Retrieval formulas will be composed of set union and intersection operations applied to *GetTerms* sets. Methods for efficiently computing the *GetTerms* sets and their unions and intersections are described in Section 4.

$$\begin{aligned}
GetVariants(p, x) &= GetTerms(p, *) \\
GetVariants(p, a) &= GetTerms(p, a) \\
GetVariants(p, f(t_1, \ldots, t_n)) &= GetTerms(p, f) \cap \\
&\quad GetVariants(p \cdot 1, t_1) \cap \cdots \cap \\
&\quad GetVariants(p \cdot n, t_n)
\end{aligned}$$

$$\begin{aligned}
GetInstances(p, x) &= Terms^4 \\
GetInstances(p, a) &= GetTerms(p, a) \\
GetInstances(p, f(t_1, \ldots, t_n)) &= GetTerms(p, f) \cap \\
&\quad GetInstances(p \cdot 1, t_1) \cap \cdots \cap \\
&\quad GetInstances(p \cdot n, t_n)
\end{aligned}$$

[2] These indexing methods ignore the identity of variables, so all are replaced by $*$.

[3] Equal terms can be retrieved by retrieving all variants and checking them for equality. If there is sufficient need for equal retrievals, each term can be stored twice, the second time with its variables indexed as if they were constants. Equal terms can then be found by retrieving variants of the term in the retrieval request, with variables again treated as if they were constants.

$$\begin{aligned}
GetGeneralizations(p,x) &= GetTerms(p,*)\\
GetGeneralizations(p,a) &= GetTerms(p,*) \cup GetTerms(p,a)\\
GetGeneralizations(p,f(t_1,\ldots,t_n)) &= GetTerms(p,*) \cup\\
&\quad (GetTerms(p,f) \cap\\
&\quad\; GetGeneralizations(p\cdot 1,t_1) \cap \cdots \cap\\
&\quad\; GetGeneralizations(p\cdot n,t_n))
\end{aligned}$$

$$\begin{aligned}
GetUnifiables(p,x) &= Terms\\
GetUnifiables(p,a) &= GetTerms(p,*) \cup GetTerms(p,a)\\
GetUnifiables(p,f(t_1,\ldots,t_n)) &= GetTerms(p,*) \cup\\
&\quad (GetTerms(p,f) \cap\\
&\quad\; GetUnifiables(p\cdot 1,t_1) \cap \cdots \cap\\
&\quad\; GetUnifiables(p\cdot n,t_n))
\end{aligned}$$

For example, instances of the term $f(a,g(b,x,h(y,z)))$ can be retrieved by

$$\begin{array}{c}
GetTerms(\langle\rangle,f)\cap\\
GetTerms(\langle 1\rangle,a)\cap\\
GetTerms(\langle 2\rangle,g)\cap\\
GetTerms(\langle 2,1\rangle,b)\cap\\
GetTerms(\langle 2,3\rangle,h)
\end{array} \tag{1}$$

and terms unifiable with it can be retrieved by

$$GetTerms(\langle\rangle,*) \cup \left[\begin{array}{c}
GetTerms(\langle\rangle,f)\\
\cap\\
GetTerms(\langle 1\rangle,*) \cup GetTerms(\langle 1\rangle,a)\\
\cap\\
GetTerms(\langle 2\rangle,*) \cup \left[\begin{array}{c}
GetTerms(\langle 2\rangle,g)\\
\cap\\
GetTerms(\langle 2,1\rangle,*) \cup GetTerms(\langle 2,1\rangle,b)\\
\cap\\
GetTerms(\langle 2,3\rangle,*) \cup GetTerms(\langle 2,3\rangle,h)
\end{array}\right]
\end{array}\right] \tag{2}$$

[4] Occurrences of *Terms* are effectively ignored, since the retrieval formula $Terms \cap X$ can be simplified to $X$.

## 3 Path Indexing

While coordinate indexing certainly yields the correct result, each *GetTerms* may contain many irrelevant terms. In particular, $GetTerms(\langle\rangle, f)$ is a set of all the terms whose top function symbol is $f$, regardless of the arguments, and $GetTerms(\langle 1\rangle, a)$ returns a list of all terms whose first argument is $a$, regardless of the top function symbol.

Coordinate indexing describes terms by a mapping from coordinates to symbols. Path indexing describes terms by a mapping from paths to symbols. The term $t = f(a, g(b, x, h(y, z)))$ can be specified by a mapping $Symbol_t$ from the paths $\langle\rangle$, $\langle f, 1\rangle$, $\langle f, 2\rangle$, $\langle f, 2, g, 1\rangle$, $\langle f, 2, g, 2\rangle$, $\langle f, 2, g, 3\rangle$, $\langle f, 2, g, 3, h, 1\rangle$, $\langle f, 2, g, 3, h, 2\rangle$ from the root node to the nodes of the tree in Figure 1, *with function symbols included*, to the symbols in $t$:

$$\begin{aligned}
Symbol_t(\langle\rangle) &= f\\
Symbol_t(\langle f, 1\rangle) &= a\\
Symbol_t(\langle f, 2\rangle) &= g\\
Symbol_t(\langle f, 2, g, 1\rangle) &= b\\
Symbol_t(\langle f, 2, g, 2\rangle) &= *^{5}\\
Symbol_t(\langle f, 2, g, 3\rangle) &= h\\
Symbol_t(\langle f, 2, g, 3, h, 1\rangle) &= *\\
Symbol_t(\langle f, 2, g, 3, h, 2\rangle) &= *
\end{aligned}$$

$Symbol_t(\langle f, 2, g, 1\rangle) = b$ should be interpreted as saying that $t$ has the symbol $b$ as the first argument of $g$, which is the second argument of the top function symbol $f$. Thus, $GetTerms(\langle f, 2, g, 1\rangle, b)$in the path-indexing method has the same value as $GetTerms(\langle\rangle, f) \cap GetTerms(\langle 2\rangle, g) \cap GetTerms(\langle 2, 1\rangle, b)$ in the coordinate-indexing method.

Let *Terms* be the set of terms stored for possible retrieval. Then the following can be used to define the path-indexing method for retrieving those terms in *Terms* that are variants of, instances of, generalizations of, or unifiable with a term. If $t$ and *Terms* are all linear terms, i.e., do not contain repeated variables, then these retrieval operations are exact. If there are nonlinear terms, extra terms may be retrieved. The expression $p \cdot f \cdot i$

[5] These indexing methods ignore the identity of variables, so all are replaced by $*$.

denotes the path $p$ extended by the $i^{\text{th}}$ argument position of function $f$, e.g., $\langle f, 2\rangle \cdot g \cdot 3$ is $\langle f, 2, g, 3\rangle$. The cases of terms with all variable arguments $f(x_1, \ldots, x_n)$ and not all variable arguments $f(t_1, \ldots, t_n)$, where at least one $t_i$ is assumed not to be a variable, are distinguished in $GetInstances$ and $GetUnifiables$.

Let $GetTerms(p, s) = \{t \in Terms | Symbol_t(p) = s\}$. Retrieval formulas will be composed of set union and intersection operations applied to $GetTerms$ sets. Methods for efficiently computing the $GetTerms$ sets and their unions and intersections are described in Section 4.

$$\begin{aligned}
GetVariants(p, x) &= GetTerms(p, *) \\
GetVariants(p, a) &= GetTerms(p, a) \\
GetVariants(p, f(t_1, \ldots, t_n)) &= GetVariants(p \cdot f \cdot 1, t_1) \cap \cdots \cap \\
&\phantom{=} GetVariants(p \cdot f \cdot n, t_n)
\end{aligned}$$

$$\begin{aligned}
GetInstances(p, x) &= Terms^{6} \\
GetInstances(p, a) &= GetTerms(p, a) \\
GetInstances(p, f(x_1, \ldots, x_n)) &= GetTerms(p, f) \\
GetInstances(p, f(t_1, \ldots, t_n)) &= GetInstances(p \cdot f \cdot 1, t_1) \cap \cdots \cap \\
&\phantom{=} GetInstances(p \cdot f \cdot n, t_n)
\end{aligned}$$

$$\begin{aligned}
GetGeneralizations(p, x) &= GetTerms(p, *) \\
GetGeneralizations(p, a) &= GetTerms(p, *) \cup GetTerms(p, a) \\
GetGeneralizations(p, f(t_1, \ldots, t_n)) &= GetTerms(p, *) \cup \\
&\phantom{=} (GetGeneralizations(p \cdot f \cdot 1, t_1) \cap \cdots \cap \\
&\phantom{=} GetGeneralizations(p \cdot f \cdot n, t_n))
\end{aligned}$$

$$GetUnifiables(p, x) = Terms$$

[6] Occurrences of $Terms$ are effectively ignored, since the retrieval formula $Terms \cap X$ can be simplified to $X$.

$$\begin{aligned}
GetUnifiables(p,a) &= GetTerms(p,*) \cup GetTerms(p,a)\\
GetUnifiables(p,f(x_1,\ldots,x_n)) &= GetTerms(p,*) \cup GetTerms(p,f)\\
GetUnifiables(p,f(t_1,\ldots,t_n)) &= GetTerms(p,*) \cup\\
&\quad (GetUnifiables(p\cdot f\cdot 1,t_1) \cap \cdots \cap\\
&\quad\ GetUnifiables(p\cdot f\cdot n,t_n))
\end{aligned}$$

For example, instances of the term $f(a,g(b,x,h(y,z)))$ can be retrieved by

$$\begin{array}{c}
GetTerms(\langle f,1\rangle,a)\cap\\
GetTerms(\langle f,2,g,1\rangle,b)\cap\\
GetTerms(\langle f,2,g,3\rangle,h)
\end{array} \tag{3}$$

and terms unifiable with it can be retrieved by

$$GetTerms(\langle\rangle,*)\cup\left[\begin{array}{c}
GetTerms(\langle f,1\rangle,*)\cup GetTerms(\langle f,1\rangle,a)\\
\cap\\
GetTerms(\langle f,2\rangle,*)\cup\left[\begin{array}{c}
GetTerms(\langle f,2,g,1\rangle,*)\cup GetTerms(\langle f,2,g,1\rangle,b)\\
\cap\\
GetTerms(\langle f,2,g,3\rangle,*)\cup GetTerms(\langle f,2,g,3\rangle,h)
\end{array}\right]
\end{array}\right] \tag{4}$$

# 4 Implementation

An efficient implementation of coordinate or path indexing depends on (1) efficiently computing *GetTerms* sets and (2) efficiently computing unions and intersections of *GetTerms* sets.

## 4.1 Computing *GetTerms* Sets

For the fastest retrieval, the set of terms in $GetTerms(p,s)$ for coordinate or path $p$ and symbol $s$ is explicitly stored. Thus, for each term in *Terms* (the set of all terms stored for possible retrieval), a pointer to the term is stored in a *GetTerms* set for each symbol in the term.

There is a large enough number of *GetTerms* sets to make it necessary to consider how to find the $GetTerms(p,s)$ sets efficiently given $p$ and $s$.

One approach is to store $GetTerms(p,s)$ in a hash table. For example, $hash(p,s)$ could compute an integer index $n$ into array $A$ such that

$$A(n) = \{(p_i, s_i, GetTerms(p_i, s_i)) | hash(p_i, s_i) = n\}.$$

$GetTerms(p,s)$ could be found, if present, by comparing $p$ and $s$ with each $p_i$ and $s_i$ in $A(n)$.

An implementation should take account of the frequent occurrence of common initial subsequences of coordinates or paths to reduce unnecessary computation and storage costs. For example, computing $hash(\langle f,2,g,1\rangle, b)$ and $hash(\langle f,2,g,3\rangle, h)$ can share the cost of computing a hash value for the common initial subsequence $\langle f,2,g\rangle$ of the two paths. Likewise, the two paths can be stored in the hash table with structure sharing of the common initial subsequence.

A second approach is to construct a discrimination net [2] (also see Section 6.1) for keys $(p,s)$ and to find $GetTerms(p,s)$ by traversing the nodes of the discrimination net in accord with $(p,s)$.

Nodes in the discrimination net are reached from their parent nodes by integer and symbol lookup operations:

$$\begin{aligned} ilp: &\quad node \times integer \rightarrow node \\ slp: &\quad node \times symbol \rightarrow node. \end{aligned}$$

The discrimination-net node $N$ corresponding to the end of the path $p = \langle f,2,g,1\rangle$, for example, can be found by

$$N = ilp(slp(ilp(slp(N_0, f), 2), g), 1),$$

where $N_0$ is the top node of the discrimination net. The $ilp$ integer lookup operation can be implemented as an array reference operation. The $slp$ symbol lookup operation can be implemented as a hash table or association list lookup operation, or an array reference operation on an integer corresponding to the symbol. Array references on integers corresponding to the symbol provide fast constant-time symbol lookup, but can be wasteful of space if the number of symbols is large and nodes often have few successors. Association lists provide space-efficient nonconstant-time symbol lookup that is fast if the number of successors is small. Hash tables provide nearly constant-time symbol lookup with space requirements that may be between those of the other techniques.

The value of $GetTerms(p, *)$ is stored in a field of node $N$; the value of $GetTerms(p, s)$ is stored in a field of node $slp(N, s)$.

This approach eliminates the need to store $p$ and $s$ with $GetTerms(p, s)$ as in the hash table approach and eliminates the need to compare $p$ and $s$ with stored $p_i$s and $s_i$s. The absence of this comparison may make the discrimination-net approach asymptotically superior, although the hash-table approach may still perform well in practical cases.

## 4.2 Computing Unions and Intersections of *GetTerms* Sets

The set union and intersection operations specified in the retrieval formulas can always be done in time proportional to the sum of the sizes of the *GetTerms* sets if each entry has an extra mark field.

For example, Eq (3) for retrieving instances of $f(a, g(b, x, h(y, z)))$ by path indexing can be computed by the algorithm

1. Mark with 1 every entry in $GetTerms(\langle f, 1\rangle, a)$.
2. Mark with 2 every entry in $GetTerms(\langle f, 2, g, 1\rangle, b)$ that is now marked with 1.
3. Retrieve every entry in $GetTerms(\langle f, 2, g, 3\rangle, h)$ that is now marked with 2.

Equation (4) for retrieving terms unifiable with it can be computed by the algorithm

1. Mark with 1 every entry in $GetTerms(\langle f, 1\rangle, a)$ or $GetTerms(\langle f, 1\rangle, *)$.
2. Mark with 2 every entry in $GetTerms(\langle f, 2, g, 1\rangle, b)$ or $GetTerms(\langle f, 2, g, 1\rangle, *)$ that is now marked with 1.
3. Retrieve every entry in $GetTerms(\langle f, 2, g, 3\rangle, h)$ or $GetTerms(\langle f, 2, g, 3\rangle, *)$ that is now marked with 2.
4. Retrieve every entry in $GetTerms(\langle f, 2\rangle, *)$ that is now marked with 1.
5. Retrieve every entry in $GetTerms(\langle\rangle, *)$.

# 5 Comparison of Coordinate and Path Indexing

Coordinate and path indexing return identical results, but path indexing can be expected to result in substantially faster retrieval than coordinate indexing.

The price for this increase in speed is a modest increase in storage cost. The same number of pointers to terms are stored in both methods, but they are divided into a larger number of *GetTerms* sets in path indexing than in coordinate indexing. The extra storage cost comes from the extra indexing structure required to distinguish among all the paths of the terms instead of all the coordinates of the terms.

Path indexing appears to be more feasible for storing and retrieving unordered collections of terms.

## 5.1 Retrieval Time

Inspection of the definitions of *GetVariants*, *GetInstances*, *GetGeneralizations*, and *GetUnifiables* for coordinate and path indexing immediately reveals the reason for the expected superiority of path indexing's retrieval time. Path indexing consistently performs set union and intersection operations on sets of terms that can reasonably be expected to be much smaller than those in coordinate indexing.

Consider Eq. (1) and Eq. (3) , which describe the retrieval of instances of $f(a, g(b, x, h(y, z)))$ by the two methods. Path indexing computes the intersection of the sets

$$\begin{aligned} GetTerms(\langle f, 1\rangle, a) \\ GetTerms(\langle f, 2, g, 1\rangle, b) \\ GetTerms(\langle f, 2, g, 3\rangle, h) \end{aligned}$$

whereas coordinate indexing computes the intersection of the probably larger, and certainly not smaller, sets

$$\begin{aligned} GetTerms(\langle 1\rangle, a) \\ GetTerms(\langle 2, 1\rangle, b) \\ GetTerms(\langle 2, 3\rangle, h) \end{aligned}$$

which must still be intersected with the sets

$$GetTerms(\langle\rangle, f)$$
$$GetTerms(\langle 2\rangle, g)$$

Comparison of Eq. (2) and Eq. (4) for retrieving terms that are unifiable with $f(a, g(b, x, h(y, z)))$ also demonstrates the expected superiority of path indexing's retrieval time.

The following table shows the number of *GetTerms* in the retrieval formula for coordinate indexing and path indexing. Only in extreme cases (no function symbols or no function symbols with nonvariable arguments) does path indexing require as many *GetTerms* as coordinate indexing, and it never requires more.

| Number of *GetTerms* in Retrieval Formula | | |
|---|---|---|
| | Coordinate | Path |
| Retrieval Type | Indexing | Indexing |
| Variant | $V+C+F$ | $V+C$ |
| Instance | $C+F$ | $C+F_V$ |
| Generalization | $V+2C+2F$ | $V+2C+F$ |
| Unifiable | $2C+2F$ | $2C+F+F_V$ |

$V$ = number of variable-symbol occurrences in retrieval term
$C$ = number of constant-symbol occurrences in retrieval term
$F$ = number of function-symbol occurrences in retrieval term
$F_V$ = number of function-symbol occurrences with only variable arguments in retrieval term

The minimum number of symbol and integer lookups (*slp* and *ilp* operations) required is the same for the two procedures:

- $C+F$ symbol lookups.
- $V+C+F-1$ integer lookups for variant and generalization retrievals.
- $C+F-1$ integer lookups for instance and unifiable retrievals.

The preceding is actually a worst-case analysis. Retrieval of some *GetTerms* sets and some symbol and integer lookup operations can be eliminated if, for example, a symbol lookup has no resulting node. This can happen because the discrimination net may contain only those nodes necessary to index the terms actually stored, not all those that might be used in retrieval requests. The more detailed indexing by function symbol and argument position of path indexing, as compared with indexing by argument position only in coordinate indexing, makes this elimination of effort more likely in path indexing than in coordinate indexing. This further enhances the superiority of path indexing.

The smaller size of the *GetTerms* sets for path indexing is a major contributor to the method's superiority over coordinate indexing. However, the magnitude of the size reduction of *GetTerms* sets depends on the stored terms themselves, so we cannot present a formal comparison.

## 5.2 Storage Cost

Storage cost is modestly greater for path indexing than for coordinate indexing.

We expect each $GetTerms(p, s)$, the set of all terms with symbol $s$ at coordinate or end of path $p$, to be stored explicitly. For each symbol occurrence in a stored term $t$, a pointer to $t$ will be stored in one of these *GetTerms* sets. Thus, for both coordinate and path indexing, the total number of pointers to stored terms, or the total size of the *GetTerms* sets, is the just the sum of the number of symbols of all the stored terms.

Although pointers to terms are stored in exactly the same number of *GetTerms* sets, there are more sets for path indexing than for coordinate indexing. Thus, the number of nodes in the discrimination net used to locate *GetTerms* sets is greater for path indexing than coordinate indexing. The number of nodes is related to the number of possible coordinates in coordinate indexing and to the number of possible paths in path indexing.

For example, consider the following rewrites which form a complete set of reductions for free groups. When they are used to simplify terms, their left-hand sides must be indexed for retrieval:

$$\begin{aligned} f(e,x) &\rightarrow x \\ f(x,e) &\rightarrow x \\ f(g(x),x) &\rightarrow e \end{aligned}$$

$$\begin{aligned}
f(x, g(x)) &\rightarrow e \\
f(f(x, y), z) &\rightarrow f(x, f(y, z)) \\
g(e) &\rightarrow e \\
g(g(x)) &\rightarrow x \\
f(g(x), f(x, y)) &\rightarrow y \\
f(x, f(g(x), y)) &\rightarrow y \\
g(f(x, y)) &\rightarrow f(g(y), g(x))
\end{aligned}$$

The size of *GetTerms* sets for the stored left-hand sides is shown below.

| Coordinate Indexing | Path Indexing |
|---|---|
| $\lvert GetTerms(\langle\rangle, f)\rvert = 7$ | $\lvert GetTerms(\langle\rangle, f)\rvert = 7$ |
| $\lvert GetTerms(\langle\rangle, g)\rvert = 3$ | $\lvert GetTerms(\langle\rangle, g)\rvert = 3$ |
| $\lvert GetTerms(\langle 1\rangle, *)\rvert = 3$ | $\lvert GetTerms(\langle f, 1\rangle, *)\rvert = 3$ |
| $\lvert GetTerms(\langle 1\rangle, e)\rvert = 2$ | $\lvert GetTerms(\langle f, 1\rangle, e)\rvert = 1$<br>$\lvert GetTerms(\langle g, 1\rangle, e)\rvert = 1$ |
| $\lvert GetTerms(\langle 1\rangle, f)\rvert = 2$ | $\lvert GetTerms(\langle f, 1\rangle, f)\rvert = 1$<br>$\lvert GetTerms(\langle g, 1\rangle, f)\rvert = 1$ |
| $\lvert GetTerms(\langle 1\rangle, g)\rvert = 3$ | $\lvert GetTerms(\langle f, 1\rangle, g)\rvert = 2$<br>$\lvert GetTerms(\langle g, 1\rangle, g)\rvert = 1$ |
| $\lvert GetTerms(\langle 2\rangle, *)\rvert = 3$ | $\lvert GetTerms(\langle f, 2\rangle, *)\rvert = 3$ |
| $\lvert GetTerms(\langle 2\rangle, e)\rvert = 1$ | $\lvert GetTerms(\langle f, 2\rangle, e)\rvert = 1$ |
| $\lvert GetTerms(\langle 2\rangle, f)\rvert = 2$ | $\lvert GetTerms(\langle f, 2\rangle, f)\rvert = 2$ |
| $\lvert GetTerms(\langle 2\rangle, g)\rvert = 1$ | $\lvert GetTerms(\langle f, 2\rangle, g)\rvert = 1$ |
| $\lvert GetTerms(\langle 1, 1\rangle, *)\rvert = 5$ | $\lvert GetTerms(\langle f, 1, f, 1\rangle, *)\rvert = 1$<br>$\lvert GetTerms(\langle f, 1, g, 1\rangle, *)\rvert = 2$<br>$\lvert GetTerms(\langle g, 1, f, 1\rangle, *)\rvert = 1$<br>$\lvert GetTerms(\langle g, 1, g, 1\rangle, *)\rvert = 1$ |
| $\lvert GetTerms(\langle 1, 2\rangle, *)\rvert = 2$ | $\lvert GetTerms(\langle f, 1, f, 2\rangle, *)\rvert = 1$<br>$\lvert GetTerms(\langle, g, 1, f, 2\rangle, *)\rvert = 1$ |
| $\lvert GetTerms(\langle 2, 1\rangle, *)\rvert = 2$ | $\lvert GetTerms(\langle, f, 2, f, 1\rangle, *)\rvert = 1$<br>$\lvert GetTerms(\langle, f, 2, g, 1\rangle, *)\rvert = 1$ |
| $\lvert GetTerms(\langle 2, 1\rangle, g)\rvert = 1$ | $\lvert GetTerms(\langle, f, 2, f, 1\rangle, g)\rvert = 1$ |
| $\lvert GetTerms(\langle 2, 2\rangle, *)\rvert = 2$ | $\lvert GetTerms(\langle, f, 2, f, 2\rangle, *)\rvert = 2$ |
| $\lvert GetTerms(\langle 2, 1, 1\rangle, *)\rvert = 1$ | $\lvert GetTerms(\langle, f, 2, f, 1, g, 1\rangle, *)\rvert = 1$ |

The indexing structure for these terms requires 24 nodes for coordinate indexing and 38 nodes for path indexing. For coordinate indexing, each of 16 *GetTerms* sets is stored in a single node and 8 additional nodes represent coordinates: $\langle\rangle$, $\langle 1\rangle$, $\langle 2\rangle$, $\langle 1, 1\rangle$, $\langle 1, 2\rangle$, $\langle 2, 1\rangle$, $\langle 2, 2\rangle$, $\langle 2, 1, 1\rangle$. For path indexing, each of 24 *GetTerms* sets is stored in a single node and 14 additional nodes represent partial paths: $\langle\rangle$, $\langle f, 1\rangle$, $\langle f, 2\rangle$, $\langle g, 1\rangle$, $\langle f, 1, f, 1\rangle$, $\langle f, 1, f, 2\rangle$, $\langle f, 1, g, 1\rangle$, $\langle f, 2, f, 1\rangle$, $\langle f, 2, f, 2\rangle$, $\langle f, 2, g, 1\rangle$, $\langle g, 1, f, 1\rangle$, $\langle g, 1, f, 2\rangle$, $\langle g, 1, g, 1\rangle$, $\langle f, 2, f, 1, g, 1\rangle$.

An optimization that reduces the length of paths and number of nodes for path indexing follows from the observation that if $g$ is a unary function, the path $\langle g, 1, f, 1\rangle$ could instead be given as $\langle g, f, 1\rangle$ because an argument of $g$ is always argument number 1. In a further optimization, 1 can always be omitted, so that $\langle f, 1, g, 1\rangle$ and $\langle f, 2, g, 1\rangle$, which denote the first argument position of $g$ in the first argument position of $f$ and the first argument position of $g$ in the second argument position of $f$, would instead be written as $\langle f, g\rangle$ and $\langle f, 2, g\rangle$.

### 5.3 Application to Unordered Terms

Path indexing's retrieval of smaller sets that more precisely match the term in the retrieval request makes it more feasible to change the definition of path to allow retrieval of unordered collections of terms.

Suppose, for example, that $j$ is a commutative function. If the term $t = j(a, b)$ is stored, we would want to retrieve it if seeking variants of $j(b, a)$ or instances of $j(x, a)$. Although in this case it would be easy to retrieve variants of both $j(a, b)$ and $j(b, a)$ or instances of both $j(a, x)$ and $j(x, a)$, this would be more complex and costly in cases where the terms have more than one function with unordered arguments.

Storage of $j(a, b)$ would ordinarily be done in path indexing according to the relation

$$\begin{aligned} \mathit{Symbol}_t(\langle\rangle) &= j \\ \mathit{Symbol}_t(\langle j, 1\rangle) &= a \\ \mathit{Symbol}_t(\langle j, 2\rangle) &= b \end{aligned}$$

Thus, pointers to $t$ would be stored in the sets $\mathit{GetTerms}(\langle\rangle, j)$, $\mathit{GetTerms}(\langle j, 1\rangle, a)$, and $\mathit{GetTerms}(\langle j, 2\rangle, b)$.

In a refinement of path indexing for unordered terms, $j(a, b)$ could be stored according to the relation

$$\begin{aligned} \mathit{Symbol}_t(\langle\rangle) &= j \\ \mathit{Symbol}_t(\langle j, ?\rangle) &= a \\ \mathit{Symbol}_t(\langle j, ?\rangle) &= b \end{aligned}$$

where $Symbol_t(\langle j, ?\rangle) = a$ specifies that $a$ occurs as an argument of top function symbol $j$. Thus, pointers to $t$ would be stored in the sets $GetTerms(\langle\rangle, j)$, $GetTerms(\langle j, ?\rangle, a)$, and $GetTerms(\langle j, ?\rangle, b)$.

The $GetVariants$, $GetInstances$, $GetGeneralizations$, and $GetUnifiables$ retrieval formulas are likewise modified to substitute ? for argument indices in the case of unordered arguments.

This is more feasible in path indexing than coordinate indexing. Doing likewise in coordinate indexing would result in pointers to $t$ being stored in $GetTerms(\langle\rangle, j)$ $GetTerms(\langle ?\rangle, a)$, and $GetTerms(\langle ?\rangle, b)$. But sets like $GetTerms(\langle ?\rangle, a)$ appear to be too undiscriminating in their membership to be practically useful, since they contain all terms with unordered argument $a$ regardless of function symbol.

# 6 Comparison with Other Indexing Methods

## 6.1 Discrimination Net

In refining the coordinate-indexing method to obtain the path-indexing method, we made retrieval more sensitive to context. We would retrieve $GetTerms(\langle f, 1\rangle, a)$, the set of all terms with $a$ as the first argument of top function symbol $f$, instead of $GetTerms(\langle 1\rangle, a)$, the set of all terms with $a$ as the first argument regardless of function symbol.

Retrieval can be made even more sensitive to context. To make retrieval sensitive to all symbols to the left of the current one, the term $t = f(a, g(b, x, h(y, z)))$ can be specified by a mapping $Symbol_t$ of the tree in Figure 1 to the symbols in $t$:

$$\begin{aligned}
Symbol_t(\langle\rangle) &= f \\
Symbol_t(\langle f\rangle) &= a \\
Symbol_t(\langle f, a\rangle) &= g \\
Symbol_t(\langle f, a, g\rangle) &= b \\
Symbol_t(\langle f, a, g, b\rangle) &= *^7 \\
Symbol_t(\langle f, a, g, b, *\rangle) &= h \\
Symbol_t(\langle f, a, g, b, *, h\rangle) &= * \\
Symbol_t(\langle f, a, g, b, *, h, *\rangle) &= *
\end{aligned}$$

This mapping supports the use of a *discrimination net* or *trie* [2,3,6]. A discrimination net can be used to store the strings of symbols obtained by the preorder traversal of terms to be indexed.

Nodes in the discrimination net are reached from their parent nodes by the symbol lookup operation:[8]

$$slp: \ \ node \times symbol \rightarrow node.$$

The *slp* symbol lookup operation can be implemented in a variety of ways as discussed in Section 4.1. Terms can be stored in the *terms* field of nodes. Let $N_0$ be the top node of the discrimination net.

Let $Terms$ be the set of terms stored for possible retrieval. For each term $t$ in $Terms$ with preorder traversal $\langle s_1, \ldots, s_n \rangle$ (e.g., $\langle f, a, g, b, *, h, *, * \rangle$ for $t = f(a, g(b, x, h(y, z)))$),

$$t \in terms(slp(\cdots(slp(N_0, s_1), \cdots), s_n)).$$

The following can be used to define methods for retrieving those terms in $Terms$ that are variants of, instances of, generalizations of, or unifiable with a term. $GetVariants(N_0, \langle t \rangle)$ returns a subset of $Terms$ that includes all variants of $t$; $GetInstances(N_0, \langle t \rangle)$ returns a subset that includes all instances; $GetGeneralizations(N_0, \langle t \rangle)$ returns a subset that includes all generalizations; $GetUnifiables(N_0, \langle t \rangle)$ returns a subset that includes all terms unifiable with $t$. If $t$ and $Terms$ are all linear terms, then these retrieval operations are exact. If there are nonlinear terms, extra terms may be retrieved.

$$\begin{aligned}
GetVariants(N, \langle\rangle) &= terms(N) \\
GetVariants(N, \langle x, t_2, \ldots, t_m \rangle) &= GetVariants(slp(*), \langle t_2, \ldots, t_m \rangle) \\
GetVariants(N, \langle a, t_2, \ldots, t_m \rangle) &= GetVariants(slp(a), \langle t_2, \ldots, t_m \rangle) \\
GetVariants(N, \langle f(t'_1, \ldots, t'_n), t_2, \ldots, t_m \rangle) &= GetVariants(slp(f), \langle t'_1, \ldots, t'_n, t_2, \ldots, t_m \rangle)
\end{aligned}$$

[7] As in coordinate and path indexing, we ignore identity of variables, so all are replaced by $*$. It is feasible instead to retain variable names and bind variables while traversing the discrimination net. The search for terms can then be pruned when variable bindings conflict [6].

[8] To simplify this description, we treat $*$ as any other symbol. To save lookup time, the successor node for a variable should be stored in a separate field in the node.

$$\begin{aligned}
\mathit{GetInstances}(N,\langle\rangle) &= \mathit{terms}(N)\\
\mathit{GetInstances}(N,\langle *,t_2,\ldots,t_m\rangle) &= \bigcup_{M\in \mathit{Skip}(N)} \mathit{GetInstances}(M,\langle t_2,\ldots,t_m\rangle)\\
\mathit{GetInstances}(N,\langle a,t_2,\ldots,t_m\rangle) &= \mathit{GetInstances}(\mathit{slp}(a),\langle t_2,\ldots,t_m\rangle)\\
\mathit{GetInstances}(N,\langle f(t'_1,\ldots,t'_n),t_2,\ldots,t_m\rangle) &= \mathit{GetInstances}(\mathit{slp}(f),\langle t'_1,\ldots,t'_n,t_2,\ldots,t_m\rangle)
\end{aligned}$$

$$\begin{aligned}
\mathit{GetGeneralizations}(N,\langle\rangle) &= \mathit{terms}(N)\\
\mathit{GetGeneralizations}(N,\langle x,t_2,\ldots,t_m\rangle) &= \mathit{GetGeneralizations}(\mathit{slp}(*),\langle t_2,\ldots,t_m\rangle)\\
\mathit{GetGeneralizations}(N,\langle a,t_2,\ldots,t_m\rangle) &= \mathit{GetGeneralizations}(\mathit{slp}(*),\langle t_2,\ldots,t_m\rangle)\cup\\
&\phantom{=}\ \mathit{GetGeneralizations}(\mathit{slp}(a),\langle t_2,\ldots,t_m\rangle)\\
\mathit{GetGeneralizations}(N,\langle f(t'_1,\ldots,t'_n),t_2,\ldots,t_m\rangle) &= \mathit{GetGeneralizations}(\mathit{slp}(*),\langle t_2,\ldots,t_m\rangle)\cup\\
&= \mathit{GetGeneralizations}(\mathit{slp}(f),\langle t'_1,\ldots,t'_n,t_2,\ldots,t_m\rangle)
\end{aligned}$$

$$\begin{aligned}
\mathit{GetUnifiables}(N,\langle\rangle) &= \mathit{terms}(N)\\
\mathit{GetUnifiables}(N,\langle *,t_2,\ldots,t_m\rangle) &= \bigcup_{M\in \mathit{Skip}(N)} \mathit{GetUnifiables}(M,\langle t_2,\ldots,t_m\rangle)\\
\mathit{GetUnifiables}(N,\langle a,t_2,\ldots,t_m\rangle) &= \mathit{GetUnifiables}(\mathit{slp}(*),\langle t_2,\ldots,t_m\rangle)\cup\\
&\phantom{=}\ \mathit{GetUnifiables}(\mathit{slp}(a),\langle t_2,\ldots,t_m\rangle)\\
\mathit{GetUnifiables}(N,\langle f(t'_1,\ldots,t'_n),t_2,\ldots,t_m\rangle) &= \mathit{GetUnifiables}(\mathit{slp}(*),\langle t_2,\ldots,t_m\rangle)\cup\\
&= \mathit{GetUnifiables}(\mathit{slp}(f),\langle t'_1,\ldots,t'_n,t_2,\ldots,t_m\rangle)
\end{aligned}$$

One auxiliary function is necessary. $\mathit{Skip}(N)$ is the set of nodes in the discrimination net obtained by skipping over all the descendant nodes that correspond to skipping a single term. For all symbols $s$ for which $\mathit{slp}(N,s)$ is defined, if $s$ has arity 0 ($s$ is a constant or $*$), then $\mathit{slp}(N,s)\in \mathit{Skip}(N)$; if $s$ has arity $n>0$, then $\mathit{Skip}^n(\mathit{slp}(N,s))\subseteq \mathit{Skip}(N)$.

The *Skip* function can be implemented as described or, alternatively, the value of $\mathit{Skip}(N)$ can be stored as an explicit list in node $N$. The latter should be much more efficient, since it eliminates arity computations and traversing intermediate nodes. The extra storage required for such lists is negligible. Although *Skip* lists can be long, any node can only be an element of a single *Skip* list, so the cumulative cost of *Skip* lists is one list element per node.

Unlike coordinate and path indexing, discrimination-net indexing does not use *GetTerms* sets and does not need to compute any intermediate results by set union and intersection operations. Discrimination-net indexing retrieves all terms in $terms(N)$ for all nodes $N$ in the final recursive calls of *GetVariants*, *GetInstances*, *GetGeneralizations*, and *GetUnifiables*.

In coordinate and path indexing, the number and size of the *GetTerms* sets is usually the dominant factor in retrieval time. In discrimination-net indexing, the number of symbol lookup operations is the dominant factor.

For variant retrieval, $F + C$ symbol lookup operations must be performed, where $F$ is the number of function symbols and $C$ is the number of constant symbols in the retrieval term. As noted before, $slp(*)$ operations for variables should not be performed by real symbol lookup operations and are therefore not counted here.

For generalization retrieval, the number of symbol lookup operations is only exponentially bounded by the number of symbols in the retrieval term. For example, retrieval of generalizations of the term $f(a_1, \ldots, a_n)$ with $n$-ary function symbol $f$ and constant arguments $a_1, \ldots, a_n$ may require $2^n$ symbol lookup operations (fewer if successor nodes do not always exist).

For unifiable and instance retrievals, the number of symbol lookup operations is no longer bounded by the number of symbols in the retrieval term. The number of symbol lookup operations depends in part on the size of the sets of nodes computed by $Skip(N)$, which depends on the number and structure of the stored terms.

A contrived example to demonstrate the possibility of poor behavior in discrimination-net indexing is the storage of the addition table $plus(m, n, m + n)$ for $m$ and $n$ in the range $[0, 999]$:

$$\begin{array}{lrrr}
plus( & 0, & 0, & 0) \\
 & \vdots & & \\
plus( & 0, & 999, & 999) \\
plus( & 1, & 0, & 1) \\
 & \vdots & & \\
plus( & 999, & 999, & 1998)
\end{array}$$

and the retrieval of instances of $plus(x, y, 150)$. Path indexing immediately finds the solutions in $GetTerms(\langle plus, 3\rangle, 150)$ while discrimination-net indexing must traverse essentially the entire discrimination net, whose size exceeds 1 million nodes. On the other hand, discrimination-net indexing immediately finds instances of $plus(70, 80, z)$ while path indexing intersects the 1,000-element $GetTerms(\langle plus, 1\rangle, 70)$ and $GetTerms(\langle plus, 2\rangle, 80)$ sets in time proportional to the sum of the sizes of the sets.

For several years, one of our deduction systems [9] relied on the undocumented use of discrimination-net indexing. We developed path indexing as an alternative to discrimination-net indexing to overcome some limitations of discrimination-net indexing. We had been dissatisfied with our handling of functions that are associative and/or commutative. A provision for unordered arguments in path indexing partially addresses this concern (see Section 5.3). Discrimination-net indexing can have variable storage requirements and retrieval times that are sensitive to the encoding of the input. For example, storing $father(john, bill)$ and $father(john, mary)$ requires fewer nodes then storing them with argument order reversed because they have common initial instead of common tail subsequences in their preorder traversals. Storage requirements and retrieval time in path indexing do not depend on argument order. Storage requirements for discrimination-net indexing can sometimes be excessive. Path indexing generally requires less space.

Nevertheless, discrimination-net indexing remains very competitive. Discrimination-net indexing performs no set union and intersection operations. Variant retrieval is very rapid. Generalization retrieval also often performs well, despite the absence of a polynomial bound on the number of symbol lookup operations based on the size of the retrieval term. McCune has suggested that discrimination-net indexing be used specifically for retrieving generalizations [6] and has investigated its use for other types of retrieval. Although the number of symbol lookup operations is not bounded by the size of the retrieval term for instance and unifiable retrievals, instance and unifiable retrieval can be accelerated by using lists of pointers to successor nodes for skipped terms in the discrimination net instead of traversing the skipped discrimination-net nodes. Retrieval time is still always bounded by the number of stored terms and the size of the discrimination net.

Christian's HIPER (HIgh PErformance Rewriting) system [3] for extended Knuth-Bendix completion includes very efficient code for discrimination-net indexing. Permutation of arguments in the retrieval term permits use of permutative, including commutative, but not associative, functions. Overwhelmingly more generalization than instance retrievals are usually necessary in Knuth-Bendix completion. This diminishes the impact of possibly poor behavior of discrination nets for instance retrieval. Christian's *linear term* representation is particularly efficient for use in discrimination-net retrievals because it includes the sequence of symbols in the preorder traversal of the term.

## 6.2 Codeword Indexing

Another approach to indexing terms is *codeword indexing* [8,10]. We describe here the superimposed codewrd indexing scheme of Ramamohanrao and Shepherd [8]. A superimposed codeword indexing scheme for a conventional relational database generates a descriptor for a record by ORing codewords associated with key values.

Consider the example of an `employee(Name,Position,Department,Salary)` relation. Each possible field value has an associated code word, e.g.,

| | | | |
|---|---|---|---|
| `john` | 00010 00000 10000 | `jane` | 10000 00000 00001 |
| `clerk` | 01001 00000 00000 | `manager` | 00000 01000 01000 |
| `admin` | 00010 00100 00000 | `sales` | 00100 00000 10000 |
| 22000 | 00000 00010 00001 | 32000 | 01000 00000 01000 |

A descriptor for the record `employee(john,clerk,admin,22000)` is computed by ORing the codewords for `john`, `clerk`, `admin`, and 22000:

| | |
|---|---|
| `john` | 00010 00000 10000 |
| `clerk` | 01001 00000 00000 |
| `admin` | 00010 00100 00000 |
| 22000 | 00000 00010 00001 |
| | 01011 00110 10001 |

Query descriptors, such as for `employee(_,clerk,admin,_)`, are computed by ORing the codewords for all the specified fields:

| | |
|---|---|
| clerk | 01001 00000 00000 |
| admin | 00010 00100 00000 |
| | 01011 00100 00000 |

A record might satisfy a query if the result of ANDing the descriptors is equal to the query's descriptor. False matches are possible because different terms may coincidentally have similar descriptors. Their frequency depends in part on the number of fields, the codeword length, and the distribution of 1-bits in the codewords.

In principle, the retrieval procedure ANDs each record descriptor with the query descriptor. Whenever the result is equal to the query descriptor, the record is a possible match for the query. In practice, retrieval can be made more efficient by using a bit-slice representation of all the descriptors and examining only those bit-slices corresponding to 1-bits in the query descriptor.

To handle field values that are first-order predicate calculus terms, it is necessary to extend this standard superimposed codeword scheme.

Unlike all the previously discussed forms of indexing, which can index on every symbol in a term, it is necessary in codeword indexing to specify which argument positions are indexed.

For example, suppose terms such as `p(f(a,X),d)`, `p(X,d)`, `p(f(a,b),X)`, and `p(g(c,b),X)` are of interest and only the numbered positions of `p(1(2,3),4)` are indexed.

Using the codewords

| | |
|---|---|
| a | 01001 00000 00000 |
| b | 00000 01010 00000 |
| c | 10000 00000 00010 |
| d | 00010 00100 00000 |
| f | 00010 00000 10000 |
| g | 00000 00001 00001 |

the example terms have the following descriptors obtained by ORing the codewords for the field values and adding mask bits denoting fields that contain variables

| | |
|---|---|
| p(f(a,X),d) | 01011 00100 10000 0010 |
| p(X,d) | 00010 00100 00000 1**0 |
| p(f(a,b),X) | 01011 01010 10000 0001 |
| p(g(c,b),X) | 10000 01011 00011 0001 |

The final four bits of each descriptor denote whether that field contains a variable. The value '*' for a mask bit indicates that this bit position is irrelevant for retrieval purposes; it appears in mask bits corresponding to positions 2 and 3 of the term 1(2,3) where the term is a variable and fields 2 and 3 are absent.

Now suppose terms unifiable with p(f(a,X),Y) are to be retrieved. If the descriptor bits are denoted by 1, 2, ..., 15 and the mask bits by m1, m2, m3, m4, then terms unifiable with p(f(a,X),Y) will have codewords that satisfy the formula m1 OR ((4 AND 11) AND (m2 OR (2 AND 5))). Thus, either (1) position 1 must be filled by a variable or (2) f (bits 4 and 11) must be present and either (2a) position 2 is a variable or (2b) a (bits 2 and 5) must be present. Such retrieval formulas are reminiscent of those used in coordinate indexing, but are less exact because they do not compare symbols, only bits from their descriptors, and ignore their locations in the record.

This superimposed codeword indexing scheme was devised for indexing large Prolog databases. Two obvious extensions are required for general and efficient indexing of terms in broader contexts.

The first extension is just the specification of retrieval formulas for variant, instance, and generalization terms as well as unifiable terms.

The simple ORing of codewords to form term descriptors creates a presumption that where a value appears in a term does not matter. This is sometimes true, as in the employee relation, where values of the Name, Position, Department, and Salary fields cannot appear in any other field. However, it is often necessary to distinguish between terms such as f(g(a,b),b) and f(g(b,a),a), which are assigned the same descriptor by the above procedure. The second extension is to use different codewords for a symbol depending on the position of its occurrence in the term, e.g., by rotating a symbol's codeword some number of bits depending on the position.

Codeword indexing has several advantages, notably simplicity, low storage requirements, and use of efficient logical operations on bits.

However, it has several disadvantages. Perhaps most significantly, its time complexity is $O(N)$, where $N$ is the number of stored terms, because the descriptor of each stored term must be compared with the query. Performing this computation on bit-slices reduces the amount of work (by performing operations only on bit-slices corresponding to 1-bits in the query descriptor) and may employ parallelism to do the remaining work (logical operations on bit-slices can be performed in computer-word-size chunks) but the method remains $O(N)$.[9]

False matches can be returned if there are coincidentally similar descriptors. The other methods are exact retrieval methods except in the case of nonlinear terms. They would never retrieve a term with a constant or function symbol that failed to match a constant or function symbol in the query.[10]

Codeword indexing requires specification of which argument positions are indexed. If terms of unrestricted size and structure are stored, codeword indexing will still be able to index only the specified argument positions. This results in less complete indexing than the other methods that can index every symbol in the term.

## 7 Conclusion

We have presented the path-indexing method for indexing terms. It is a refinement of the standard coordinate-indexing method that was proposed for the PLANNER AI programming language and was used in LMA for implementing deduction systems. Path indexing nearly always retrieves terms by computing unions and intersections of fewer, smaller sets of terms than coordinate indexing. Path indexing requires somewhat more space than coordinate indexing, but we expect that the substantially faster path indexing will nearly always be useable whenever coordinate indexing is.

However, the superiority of path indexing over coordinate indexing does not demonstrate that it should be the method of choice in general. We have described and partially

[9] Hierarchical codeword indexing may overcome the $O(N)$ time requirement in practice, but this involves the greater complexity of multiple indexing, initially with much wider codewords.

[10] One consequence of this feature is that a faster version of the unification algorithm that does not check whether constants and functions are identical can be used to unify the query with retrieved terms.

analyzed the behavior of two important alternative methods: discrimination-net indexing and codeword indexing. Unfortunately, formal comparison of the methods will generally fail to justify a preference for one method, since the worst-case assumptions employed by most formal analyses are unrealistic, and average-case analyses are difficult to produce and require definition of the average case.

Therefore, optimal selection of an indexing method may require experiments on sets of terms that occur in practice, with the correct proportion of variant, instance, generalization, and unifiable retrievals as well as addition and deletion operations. For example, discrimination-net indexing might be preferable for some sets of terms if instance and unifiable retrieval is much less frequent than variant and generalization retrieval. Indexing terms in more than one way (e.g., discrimination net for variant and generalization retrieval, and path indexing for instance and unifiable retrieval) may be a useful approach to getting the best performance.

We have formulated the following guidelines for selecting a method. If speed is the paramount concern, discrimination-net indexing is likely to be best, although there are exceptions to this rule. If storage requirements are an issue, or discrimination-net indexing works poorly in a particular case, coordinate or path indexing can be used. The latter offers much greater speed at the cost of some increase in space. If the amount of storage required must be minimized, codeword indexing can be used. The bit manipulation operations in codeword indexing are individually quite fast, but the method has the disadvantages that retrieval time is linear in the number of stored terms, retrieval is less exact than for the other methods, and terms cannot be indexed arbitrarily deeply.

## References

[1] Chang, C.L. and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, New York, 1973.

[2] Charniak, E., C.K. Riesbeck, and D.V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

[3] Christian, J. Fast Knuth-Bendix completion: summary. *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications,* Chapel Hill, North Carolina, April 1989, 551–555.

[4] Hewitt, C. Description and theoretical analysis (using schemata) of Planner: a language for proving theorems and manipulating models in a robot. Ph.D. dissertation, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1971.

[5] Knuth, D.E. *The Art of Computer Programming: Sorting and Searching.* Addison-Wesley, Reading, Massachusetts, 1973.

[6] McCune, B. An indexing mechanism for finding more general formulas. *Association for Automated Reasoning Newsletter,* No. 9 (January 1988), 7–8.

[7] Overbeek, R.A. and E.L. Lusk. Data structures and control architecture for implementation of theorem-proving programs. *Proceedings of the 5th International Conference on Automated Deduction,* Les Arcs, France, July 1980, 232–249.

[8] Ramamohanarao, K. and J. Shepherd. A superimposed codeword indexing scheme for very large Prolog databases. *Proceedings of the Third International Conference on Logic Programming,* London, England, 1986.

[9] Stickel, M.E. The KLAUS automated deduction system. System abstract in *Proceedings of the 8th International Conference on Automated Deduction,* Oxford, England, July 1986, 703–704.

[10] Wise, M.J. and D.M.W. Powers. Indexing Prolog clauses via superimposed code words and field encoded words. *Proceedings of the 1984 International Symposium on Logic Programming,* Atlantic City, New Jersey, February 1984, 203–210.

[11] Wos, L., R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.